School of Computing and Information Systems

comp10002 Foundations of Algorithms
Semester 2, 2022
Assignment 1

Learning Outcomes

In this assignment you will demonstrate your understanding of arrays, strings, functions, and the `typedef` facility. You must *not* make any use of `malloc()` (Chapter 10) or file operations (Chapter 11) in this project. You *may* use `struct` types (Chapter 8) if you wish, but are not required to.

Web Search

We have all used web search tools. In response to a query, a search engine result page (SERP) is generated, containing (among many other elements) a list of URLs and *snippets*, with each snippet a short extract from the corresponding web page. A typical snippet contains around 20–30 words and is selected from the corresponding document to show the context in which query terms appear.[1] For example, the query "*web search algorithms*" at a commercial web search service resulted (August 2022) in the following URLs (in blue), page titles (in italics), and snippets (with matching key words in bold) being offered:

1. `https://www.searchenginejournal.com/search-engines/algorithms/`
   How Search Engine Algorithms Work: Everything You Need to . . .
   A **search algorithm** is a massive collection of other **algorithms**, each with its own purpose and task. Here's how it all works.
2. `https://shiftweb.com/what-is-a-search-engine-algorithm`
   What is a Search Engine Algorithm and Why is it Important?
   A **search** engine **algorithm** is a complex **algorithm** used by **search** engines such as Google, Yahoo, and Bing to determine a **web** page's. . .
3. `https://www.brafton.com.au/glossary/search-algorithm/`
   Search Algorithm - Brafton
   A **search algorithm** defines the process and factors used by a **search** company to rank websites and organize their **search** ranking with an accurate response to. . .
4. `https://moz.com/beginners-guide-to-seo/how-search-engines-operate`
   How Search Engines Work: Crawling, Indexing, and Ranking
   To determine relevance, **search** engines use **algorithms**, a process or formula . . . These **algorithms** have gone through many changes over the years. . .
5. `https://www.seoquake.com/blog/how-search-engine-algorithms-work/`
   Everything You Need to Know How Search Engine Algorithms . . .
   The **search** engine **algorithm** tries to determine the nature of the information by analyzing the user's **search** intent. The different. . .

Note how the fourth snippet is actually composed of two shorter segments, indicated by the ". . ." in the middle of it. In the other four snippets the generator decided that a single contiguous segment of text was the best representation.

[1] In early search services a snippet was generated for each document at the time it was indexed, and was independent of the any query. Query-biased snippets involve rather more computation and add to the time taken to evaluate queries, but are generally agreed to be more intuitive for users to digest and react to.

Your Mission... In this assignment you will build a program that reads paragraphs of text from `stdin`, builds a snippet for each paragraph according to certain rules, and then writes those snippets to `stdout`. You can either use `Grok` and the "terminal" facility that it provides, or develop your program outside of Grok. You should make use of functions in `ctype.h` and `string.h`. There is a handout linked from the LMS that provides guidance on the use of the Grok terminal.

Submission is via upload to the LMS. *You cannot submit via Grok.* Note that to be eligible for maximum marks in each stage you must *exactly* match the required output, including the formatting.

Stage 0 – Getting Started (maximum: 0/20 marks)

Copy the skeleton program `ass1-skel.c` and sample input files from the Assignment 1 LMS page, and check that you can compile the program via either `Grok` or `gcc`. The skeleton program contains a simple `main()` function and a version of the `get_word()` function (make careful note of how it is different), and when executed it simply copies the input to the output, one word per line, sometimes including trailing punctuation. Try compiling and running the skeleton program before you do anything else: The skeleton file also includes a very important *authorship declaration*. Substantial mark penalties apply if you do not include it in your submission, or if you do not sign it. *This is an absolute requirement, and no excuses of any sort will be accepted.*

Stage 1 – Reading Text (maximum: 12/20 marks)

Now alter function `get_word()` so that it returns one of three status values: `WORD_FND` if it identified a word; `PARA_END` if instead of a word, a paragraph boundary was identified; and `EOF` if the end of the input file was reached. Paragraphs are identified in the input text by the combination `"\n\n"`, that is, two consecutive newline characters; or by reaching the end of file after one or more alphabetic words have been found since the last paragraph boundary A suitable `#define` appears in the skeleton code.

Once that is done, alter the processing loop in the `main()` so that it processes one paragraph at a time rather than one word at a time. To accomplish this step you will probably want to introduce a new function (perhaps called `get_paragraph()`) that calls `get_word()` repeatedly, to build up a paragraph of words in an array. You may assume that each paragraph has at most `MAX_PARA_LEN` words in it, each of at most `MAX_WORD_LEN` characters.

Output: The required output from this stage is a message for each paragraph that indicates the paragraph number, and its length in words. See the LMS for examples of what is required.

Stage 2 – Matching Words and Printing Text (maximum: 16/20 marks)

The second task to write an output function that writes each paragraph's words across the page, making sure that no line contains more than 72 characters. Each "word" that is output should be separated from the previous word in the same line by a single blank character. You'll need to keep track of how many bytes have been written so far in each output line, and insert a newline character (instead of a blank) if the next word needing to be written won't fit within the current line.

Then alter your program so that any words in each paragraph that match any words supplied on the command-line (use `argc` and `argv` to access these) in a case-insensitive manner (look at the library function `strncasecmp()`) are "bolded" in the output text by putting "`**`" before and after them. Be sure to handle the punctuated words properly; the punctuation should be retained, but outside the "`**`". For example, if "`web`" is supplied on the command-line as one of the query terms, then the word "`Web;`" in the input text must place "`**Web**;`" into the output text. (Yes, in a real program we'd generate HTML tags to make the selected words bold, but for simplicity we'll stay with plain text.)

Output: Examples showing the required output are linked from the LMS page.

Stage 3 – Building Snippets (maximum: 20/20 marks)

Suppose that the *score* of a snippet relative to a set of *query terms* is calculated as:

- Add $15/(s+10)$ points, where s is the start word of the snippet, counting from the first word of the paragraph as word number zero; plus
- Add $\ell/2$ points for each *different* term that appears in the snippet, where ℓ is the length of that query term; plus
- Add 1.0 points for every other repetition of any query term that appears multiple times in the snippet; plus
- Add 0.6 points if the snippet starts with the word immediately following a punctuated word (taking word "-1" to also be punctuated); plus
- Add 0.3 points if the snippet ends with a punctuated word; and then
- Subtract 0.1 points for each word that the snippet exceeds `MIN_SNIPPET_LEN` in length.

Snippets cannot be longer than `MAX_SNIPPET_LEN` words. Similarly, snippets cannot be shorter than `MIN_SNIPPET_LEN` words, except if the input paragraph is also shorter than `MIN_SNIPPET_LEN` words, in which case the whole input paragraph always becomes the snippet. If a snippet ends with a punctuated word that has a period "." or a question mark "?" or a exclamation mark "!" attached, then no "..." should be added. All other cases should have dots added immediately after the last word in the snippet, even if it takes the last output line past 72 characters.

Note that in the web search example on the first page the query term "*algorithm*" also matches against the word "*algorithms*", this is a process known as *stemming*. Whole books have been written about stemming rules for English, so that "compute", "computer", "computed", "computation", "computational", "computing", "computability", "computably", and so on, are all taken as being the same underlying word. Stemming is definitely outside the scope of this assignment, and we will just locate exact (except for case, and for trailing punctuation) matches against the query terms.

Extend your Stage 2 program so that the highest-scoring contiguous snippet is identified for each input paragraph. In the case of scores that are equal, select the snippet that starts closest to the beginning of the paragraph. And if there are *still* ties, print out the shortest legal maximum-score snippet starting at that point. An exhaustive search approach to finding the best snippet in each paragraph is perfectly acceptable, and there won't be marks allocated to efficiency (unless your code is so bad it can't run in a reasonable time on typical non-extreme paragraphs of text). You should use standard functions such as `strncasecmp()` to compare words – no need for KMP or BMH in this task.

Output: Your output for this stage should show the required maximal-score snippet for each input paragraph, with query terms highlighted. Lines should again be broken so that they are at most 72 characters long (by calling the same function, I hope). See the LMS for examples.

Stage 007 – Split Snippets (no extra marks)

Finished already? Suddenly bored?? Worried about where your next FUN might come from??? Go back and look at the fourth example snippet on the first page.

If you want to further hone your programming skills, implement the following rules for split snippets: (1) find the best snippet of length between 10 and 15 words inclusive starting at a word in the first half of the paragraph, suppose it runs from f_1 to ℓ_1; then (2) find the best snippet of length 10 to 15 words between ℓ_1+1 and the end of the paragraph, suppose it runs from f_2 to ℓ_2; then (3) if $\ell_1+1=f_2$ join them back together and print a single snippet; or (4) if $\ell+1<f_2$ print the two parts with "..." between them, including blanks on both sides, and with all of those five characters included when counting the length of the output lines. The score of the new snippet will be the sum of the two partial scores, even if they get joined back together and don't have the "..." in the middle.

Marking? This stage is *purely recreational*. Please don't submit it for marking!

Debugging...

You will probably find it helpful to include a `DEBUG` mode in your program that prints out intermediate data and variable values. Use `#if (DEBUG)` and `#endif` around such blocks of code, and then `#define DEBUG 1` or `#define DEBUG 0` at the top. Turn off the debug mode when making your final submission, but leave the debug code in place. The FAQ page has more information about this.

Boring But Important...

This project is worth 20% of your final mark, and is due at **6:00pm on Friday 16 September**. Submit your program for assessment **via the link to GradeScope at the bottom of the LMS Assignment 1 page**. Submissions cannot be made via Grok. Best advice? *Start early and to finish early!!*

Submissions that are made after the deadline will incur penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. If you attend a GP or other health care service as a result of illness, be sure to obtain a letter from them that describes your illness and their recommendation for treatment. Suitable documentation should be attached to **all** extension requests.

Multiple submissions may be made; only the last submission that you make before the deadline will be marked. If you make any late submission at all, your on-time submissions will be ignored, and if you have not been granted an extension, the late penalty will be applied.

A rubric explaining the marking expectations is linked from the LMS, and you should study it carefully. Marks and feedback will be provided approximately two weeks after submissions close.

Academic Honesty: You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone else; do **not** have any "accidents" that allow others to access your work; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" if they ask to see your program, pointing out that your "**no**", and their acceptance of that decision, are the only way to preserve your friendship. See `https://academicintegrity.unimelb.edu.au` for more information. Note also that solicitation of solutions via posts to online forums, whether or not there is payment involved, is also Academic Misconduct. In the past students have had their enrolment terminated for such behavior.

The skeleton program includes an Authorship Declaration that you must "sign" and include at the top of your submitted program. Marks will be deducted if you do not include the declaration, or do not sign it, or do not comply with its expectations. A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions. Students whose programs are identified as containing significant overlaps will have substantial mark penalties applied, or be referred to the Student Center for possible disciplinary action.

Nor should you post your code to any public location (`github`, `codeshare.io`, etc) while the assignment is active or prior to the release of the assignment marks.

And remember, Algorithms are Fun!